MythX

REPORT 5F6AC62264B93A00182349E1

Created	Wed Sep 23 2020 03:50:58 GMT+0000 (Coordinated Universal Time)
Number of analyses	1
User	kohshi.shiba@gmail.com

REPORT SUMMARY

Analyses ID	Main source file	Detected vulnerabilities
ab727016-2a79-4ed9-9a3b-b91d6a43d871	/contracts/linerv1.sol	5

Analysis ab727016-2a79-4ed9-9a3b-b91d6a43d871

MythX

Started	Wed Sep 23 2020 03:51:02 GMT+0000 (Coordinated Universal Time)
Finished	Wed Sep 23 2020 04:36:22 GMT+0000 (Coordinated Universal Time)
Mode	Deep
Client Tool	Mythx-Vscode-Extension
Main Source File	/Contracts/Linerv1.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	0	5

ISSUES

LOW SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.6.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file

/contracts/linerv1.sol

Locations

```
*/

pragma solidity ^0.6.0;
pragma experimental ABIEncoderV2;
```

LOW SWC-110

An assertion violation was triggered.

It is possible to cause an assertion violation. Note that Solidity assert() statements should only be used to check invariants. Review the transaction trace generated for this issue and either make sure your program logic is correct, or use require() instead of assert() if your goal is to constrain user inputs or enforce preconditions. Remember to validate inputs from both callers (for instance, via passed arguments) and callees (for instance, via return values).

Source file

/contracts/linerv1.sol

Locations

```
tokenValue = tokenValue.sqrt();
tokenValue -= initialPrice;
tokenValue /= priceIncrement;
tokenValue -= supply;
if (
```

LOW

SWC-116

A control flow decision is made based on The block.timestamp environment variable.

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source file

`/contracts/linerv1.sol`

Locations

```
///@dev a modifier manages market transitions
modifier marketStatusTransitions() {
if (marketStatus == MarketStatus.BeforeTrading && now >= startTime) {
_nextMarketStatus();
}
if (marketStatus == MarketStatus.Trading && now >= endTime) {
_nextMarketStatus();
```

LOW

SWC-116

A control flow decision is made based on The block.timestamp environment variable.

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source file

`/contracts/linerv1.sol`

Locations

```
_nextMarketStatus();
}
if (marketStatus == MarketStatus.Trading && now >= endTime) {
_nextMarketStatus();
}
_;
}
```

LOW

SWC-123

Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

Source file

/contracts/linerv1.sol

Locations

```
import "./IMarket.sol";

contract LinerV1 is ERC1155, IMarket {
using SafeMath for uint256;
using Sqrt for uint256;
using Address for address;

/**
* EVENTS
*/

event Buy(
address buyer,
address to,
uint256 outcomeIndex,
uint256 investValue,
uint256 returnValue
);
event Sell(
address seller,
address to,
uint256 outcomeIndex,
uint256 sellValue,
uint256 returnValue
);
event Claimed(
address owner,
uint256 outcomeIndex,
uint256 claimedValue,
uint256 returnValue
);
event FeeCollected(
address beneficiary,
uint256 totalAccrued,
uint256 collected
);
event FeeWithdrawal(address beneficiary, uint256 amount);
event MarketSettled(uint256[] report, uint256[] payout);
event MarketStatusChanged(MarketStatus statusValue);

/**
* MARKET CONSTANTS
*/

/// @dev Global denominator e.g., 1.000% = 1000 & need to be devided by 100000
uint256 private constant GLOBAL_DENOMINATOR = 100000;

/// @dev The factory address that deployed this contract
address private factory;

/// @dev True once initialized through initialize()
bool private initialized;

/// @dev Decimals for option tokens
uint8 public decimals = 18;
```

```

/// @dev The price to buy option increase as new token issued
uint256 public priceIncrement;

/// @dev The minimum amount of `currency` investment accepted.
uint256 public minInvestment;

/// @dev The minimum amount of `currency` investment accepted.
uint256 public startPrice;

/// @dev When the sell option is disabled, option tokens cannot be sold. (0 = true)
uint256 public sellOption;

///@dev Market contents (Registered when market is created)
bytes32 public hashID;
uint256 public startTime;
uint256 public endTime;
uint256 public reportTime;
address public oracle;
IERC20 public token;
address[] public beneficiaries;
uint256[] public shares;

/**
* OUTCOMES
*/
struct Outcome {
uint256 supply;
uint256 reserve;
uint256 dividend;
}
///@dev mapping for each outcome
mapping(uint256 => Outcome) public outcome;

///@dev total number of outcomes
uint256 public outcomeNumbers;

/**
* MARKET VARIABLES
*/

/// @dev collected fee balances
mapping(address => uint256) public collectedFees;

/**
* MARKET STATES MANAGEMENT
*/

///@dev Market status transition management
enum MarketStatus {BeforeTrading, Trading, Reporting, Finalized}
MarketStatus public marketStatus;

///@dev a modifier checks the current market status
modifier atMarketStatus(MarketStatus _marketStatus) {
require(marketStatus == _marketStatus);
_;
}

///@dev a modifier manages market transitions
modifier marketStatusTransitions() {
if (marketStatus == MarketStatus.BeforeTrading && now >= startTime) {
_nextMarketStatus();
}
```

```
if (marketStatus == MarketStatus.Trading && now >= endTime) {
_nextMarketStatus();
}
_;
}

constructor() public {}

/**
* @dev Validate market
* This function validates the argument set for initialization.
* Can be called before contract deployments.
* question = the thesis of the prediction market
* outcomes = potential outcomes
* conditions[0] = startTime
* conditions[1] = endTime
* conditions[2] = reportTime
* conditions[3] = priceIncrement
* conditions[4] = sell option (0:yes 1:no)
* conditions[5] = minimum investment value
* conditions[6] = start price
* references[0] = ERC20 token used as the collateral
* references[1] = oracle address settles the market
* beneficiaries[] = beneficiary addresses collect fees
* shares[] = fee shares
* detail = any additiona info about the market
*/
function validate(
string memory _settings,
uint256 _outcomeNum,
uint256[] memory _conditions,
address[] memory _references,
address[] memory _beneficiaries,
uint256[] memory _shares
) public override view returns (bool) {
require(bytes(_settings).length > 10);
require(_outcomeNum >= 2 && _outcomeNum <= 10);
require(
_conditions[1].sub(_conditions[0]) > 1 days &&
_conditions[1].sub(now) > 1 days &&
_conditions[2] >= _conditions[1]
);
require(_conditions[3] > 0);
require(_conditions[4] < 2);
require(_references[0] != address(0) && _references[1] != address(0));
require(_beneficiaries.length <= 3);
require(_beneficiaries.length == _shares.length);

uint256 share;
for (uint256 i = 0; i < _shares.length; i++)
share = share.add(_shares[i]);
require(GLOBAL_DENOMINATOR > share);

return true;
}

/**
* @dev Initialize market
* This function registers market conditions.
* arguments are verified by the 'validate' function.
*/
function initialize(
string memory _settings,
uint256 _outcomeNum,
uint256[] memory _conditions,
address[] memory _references,
address[] memory _beneficiaries,
uint256[] memory _shares
) public override returns (bool) {
require(
validate(
_settings,
_outcomeNum,
_conditions,
_references,
_beneficiaries,
_shares
)
);

require(initialized == false);
initialized = true;
outcomeNumbers = _outcomeNum;
startTime = _conditions[0];
endTime = _conditions[1];
reportTime = _conditions[2];
priceIncrement = _conditions[3];
sellOption = _conditions[4];
minInvestment = _conditions[5];
startPrice = _conditions[6];
token = IERC20(_references[0]);
oracle = _references[1];
beneficiaries = _beneficiaries;
shares = _shares;
factory = msg.sender;

hashID = keccak256(
abi.encodePacked(
_settings,
_outcomeNum,
_conditions,
_references,
_beneficiaries,
_shares
)
);

marketStatus = MarketStatus.BeforeTrading;
return true;
}

/**
* @dev Buy
* Market participants can buy option tokens through this function.
* _params[0] investmentAmount,
* _params[1] minTokensBought,
* _params[2] outcomeIndex,
* _params[3] fee,
* _addresses[0] owner,
* _addresses[1] to
* _addresses[2] beneficiary,
*/
function buy(uint256[] memory _params, address[] memory _addresses)
public
marketStatusTransitions
atMarketStatus(MarketStatus.Trading)
```

```
{
require(_params[1] > 0, "MUST_BUY_AT_LEAST_1");

// Calculate the tokenValue for this investment
uint256 tokenValue = calcBuyAmount(_params[0], _params[2], _params[3]);
require(tokenValue >= _params[1], "PRICE_SLIPPAGE");

IERC20(token).transferFrom(msg.sender, address(this), _params[0]);
if (shares.length > 0 || _params[3] > 0) {
uint256 afterFee = _collectFees(
_params[0],
_params[3],
_addresses[2]
);
outcome[_params[2]].reserve = outcome[_params[2]].reserve.add(
afterFee
);
} else {
outcome[_params[2]].reserve = outcome[_params[2]].reserve.add(
_params[0]
);
}

_mint(_addresses[1], _params[2], tokenValue, "");
outcome[_params[2]].supply = outcome[_params[2]].supply.add(tokenValue);

emit Buy(msg.sender, _addresses[1], _params[2], _params[0], tokenValue);
}

/**
* @dev Sell
* Market participants can sell option tokens through this function.
* _params[0] sellAmount,
* _params[1] minReturned,
* _params[2] outcomeIndex,
* _addresses[0] owner,
* _addresses[1] to
*/
function sell(uint256[] memory _params, address[] memory _addresses)
public
marketStatusTransitions
atMarketStatus(MarketStatus.Trading)
{
require(
balanceOf(_addresses[0], _params[2]) >= _params[0],
"INSUFFICIENT_AMOUNT"
);
require(
msg.sender == _addresses[0] ||
_operatorApprovals[_addresses[0]][msg.sender],
"NOT_ELIGIBLE_TO_SELL"
);
uint256 returnValue = calcSellAmount(_params[0], _params[2]);
require(returnValue >= _params[1], "PRICE_SLIPPAGE");
_burn(_addresses[0], _params[2], _params[0]);
outcome[_params[2]].reserve = outcome[_params[2]].reserve.sub(
returnValue
);
outcome[_params[2]].supply = outcome[_params[2]].supply.sub(_params[0]);
IERC20(token).transfer(_addresses[1], returnValue);
emit Sell(
msg.sender,
_addresses[0],
_params[2],
_params[0],
returnValue
);
}

/**
* @dev Settle
* Registered oracle settles market by reporting payout shares.
*/
function settle(uint256[] memory report)
public
marketStatusTransitions
atMarketStatus(MarketStatus.Reporting)
{
require(msg.sender == oracle, "UNAUTHORIZED_ORACLE");

uint256 total;
for (uint256 i = 0; i < report.length; i++) {
total = total.add(report[i]);
}
require(
total == GLOBAL_DENOMINATOR && report.length == outcomeNumbers,
"INVALID_REPORT"
);
_nextMarketStatus();

/**
* If there is no supply for a winning option,
* the dividend of that will be distributed to all token holders.
*/
uint256 totalReserve;
uint256 totalSupply;
uint256 bonus;
uint256[] memory _payout = new uint256[](outcomeNumbers);
for (uint256 i = 0; i < outcomeNumbers; i++) {
totalReserve = totalReserve.add(outcome[i].reserve);
totalSupply = totalSupply.add(outcome[i].supply);
}
for (uint256 i = 0; i < outcomeNumbers; i++) {
if (outcome[i].supply == 0) {
uint256 temp = BigDiv.bigDiv2x1(
totalReserve,
report[i],
GLOBAL_DENOMINATOR
);
bonus = bonus.add(temp);
}
}
for (uint256 i = 0; i < report.length; i++) {
if (bonus > 0) {
if (outcome[i].supply != 0) {
uint256 allocation = BigDiv.bigDiv2x1(
totalReserve,
report[i],
GLOBAL_DENOMINATOR
);
uint256 bonusShare = BigDiv.bigDiv2x1(
bonus,
outcome[i].supply,
totalSupply
);
outcome[i].dividend = bonusShare + allocation;
_payout[i] = bonusShare + allocation;
}
} else {
uint256 allocation = BigDiv.bigDiv2x1(
totalReserve,
report[i],
GLOBAL_DENOMINATOR
);
outcome[i].dividend = allocation;
_payout[i] = allocation;
}
}

emit MarketSettled(report, _payout);
}

/**
* @dev Winnig token holders can claim redemption through this function
*/
function claim(address account)
public
atMarketStatus(MarketStatus.Finalized)
{
uint256 redemption;
for (uint256 i = 0; i < outcomeNumbers; i++) {
uint256 balance = balanceOf(account, i);
if (balance > 0) {
if (outcome[i].dividend > 0) {
uint256 value = BigDiv.bigDiv2x1(
outcome[i].dividend,
balance,
outcome[i].supply
);
_burn(account, i, balance);
outcome[i].supply = outcome[i].supply.sub(balance);
outcome[i].dividend = outcome[i].dividend.sub(value);
redemption = redemption.add(value);
emit Claimed(account, i, balance, value);
}
}
}
if (redemption > 0) {
IERC20(token).transfer(account, redemption);
}
}

/**
* @dev Beneficiaries can withdraw fees through this function.
*/
function withdrawFees(address account) public {
uint256 amount = collectedFees[account];
collectedFees[account] = 0;
emit FeeWithdrawal(account, amount);
IERC20(token).transfer(account, amount);
}

/**
* @dev Calclate estimate option token amount for the investment at a time.
*/
function calcBuyAmount(
uint256 investmentAmount,
uint256 outcomeIndex,
uint256 fee
```

```
) public view returns (uint256) {
if (investmentAmount < minInvestment) {
return 0;
}

/**
* Calculate the fee rate for this investment.
*/

uint256 afterFee;
if (shares.length > 0 || fee > 0) {
uint256 feeRate;
for (uint256 i = 0; i < shares.length; i++) {
feeRate = feeRate.add(shares[i]);
}
feeRate = feeRate.add(fee);
require(feeRate < GLOBAL_DENOMINATOR);
uint256 fees = BigDiv.bigDiv2x1(
investmentAmount,
feeRate,
GLOBAL_DENOMINATOR
);
afterFee = investmentAmount.sub(fees);
} else {
afterFee = investmentAmount;
}

/**
* Calculate the tokenValue for this investment.
*/
uint256 supply = outcome[outcomeIndex].supply;
uint256 reserve = outcome[outcomeIndex].reserve;
uint256 newReserve = reserve + afterFee;
uint256 initialPrice = startPrice * 1e18;
uint256 tokenValue = ((2 *
(priceIncrement.mul(1e18)).mul(newReserve.mul(1e18))) +
(initialPrice**2));
tokenValue = tokenValue.sqrt();
tokenValue -= initialPrice;
tokenValue /= priceIncrement;
tokenValue -= supply;
if (
marketStatus == MarketStatus.Trading ||
marketStatus == MarketStatus.BeforeTrading
) {
return tokenValue;
} else {
return 0;
}
}

/**
* @dev Calclate estimate collateralize token value for selling ptions
*/
function calcSellAmount(uint256 sellAmount, uint256 outcomeIndex)
public
view
returns (uint256)
{
require(sellOption == 0, "SELL_OPTION_IS_DISABLED");
if (marketStatus == MarketStatus.Trading) {
uint256 supply = outcome[outcomeIndex].supply;
uint256 reserve = outcome[outcomeIndex].reserve;
```

```
require(supply >= sellAmount, "BEYOND_SUPPLY");
if (supply == 0) {
return 0;
}
/**
* Calculate the token return for this reserve token sale.
*/
uint256 supplyAfter = supply.sub(sellAmount);
if (supplyAfter == 0) {
return reserve;
} else {
uint256 price = BigDiv
.bigDiv2x1(supplyAfter, priceIncrement, 1e18)
.add(startPrice);
uint256 reserveAfter = BigDiv
.bigDiv2x1(price.add(startPrice), supplyAfter, 1e18)
.div(2);
uint256 retVal = reserve - reserveAfter;
return retVal;
}
} else {
return 0;
}
}

/**
* @dev function to get pool balance for each option
*/
function getStake(uint256 outcomeIndex) public view returns (uint256) {
return outcome[outcomeIndex].reserve;
}

/**
* @dev function to get supply for each option
*/
function getSupply(uint256 outcomeIndex) public view returns (uint256) {
return outcome[outcomeIndex].supply;
}

/**
* @dev a function to check the factory address
*/
function creator() public override view returns (address) {
return factory;
}

/**
* @dev Validate market question and outcome lists
*/
function validateHash(
string memory _settings,
uint256 _outcomeNum,
uint256[] memory _conditions,
address[] memory _references,
address[] memory _beneficiaries,
uint256[] memory _shares
) public override view returns (bool) {
bytes32 hash = keccak256(
abi.encodePacked(
_settings,
_outcomeNum,
_conditions,
_references,
_beneficiaries,
_shares
)
);
return (hash == hashID);
}

function _nextMarketStatus() internal {
marketStatus = MarketStatus(uint256(marketStatus) + 1);
emit MarketStatusChanged(marketStatus);
}

function _collectFees(
uint256 amount,
uint256 fee,
address beneficiary
) internal returns (uint256) {
uint256 fees;
for (uint256 i = 0; i < shares.length; i++) {
uint256 portion = BigDiv.bigDiv2x1(
amount,
shares[i],
GLOBAL_DENOMINATOR
);
collectedFees[beneficiaries[i]] = collectedFees[beneficiaries[i]]
.add(portion);
fees = fees.add(portion);
emit FeeCollected(
beneficiaries[i],
collectedFees[beneficiaries[i]],
portion
);
}
if (fee > 0) {
uint256 portion = BigDiv.bigDiv2x1(amount, fee, GLOBAL_DENOMINATOR);
collectedFees[beneficiary] = collectedFees[beneficiary].add(
portion
);
fees = fees.add(portion);
emit FeeCollected(beneficiary, collectedFees[beneficiary], portion);
}
return amount.sub(fees);
}
}
```